Flink-Hive Metastore Connectivity Design

Xuefu Zhang

Motivation

With its great reputation in the open source community and the streaming processing industry, Flink has shown its potentials in batch processing. Improving Flink's batch processing, especially in terms of SQL, would benefit a greater adoption of Flink beyond streaming processing and offer user a complete solution for both their streaming and batch processing needs.

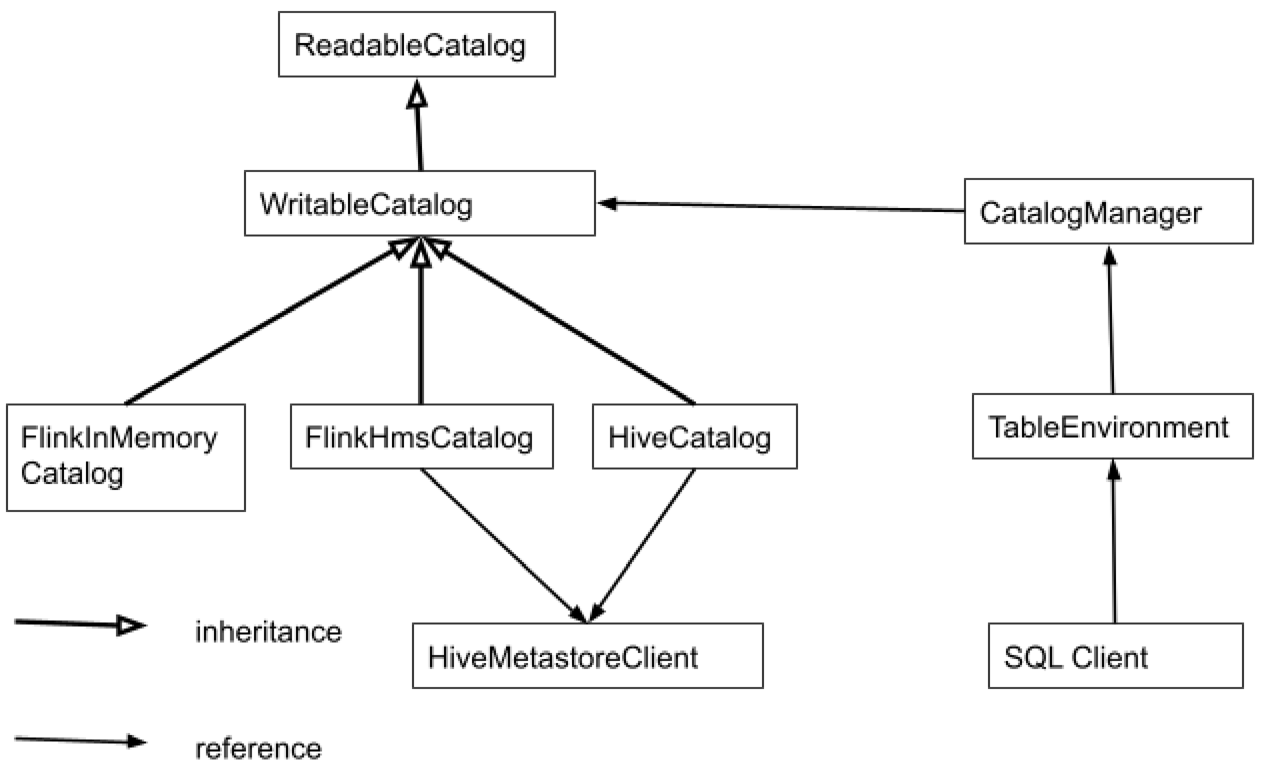
On the other hand, Hive has established its focal point in big data technology and its complete ecosystem. For most of big data users, Hive is not only a SQL engine for big data analytics and ETL, but also a data management platform, on which data are discovered, defined, evolved. In another words, Hive is a de facto standard for big data on Hadoop.

Therefore, it's imperative for Flink to integrate with Hive ecosystem to further its reach to batch and SQL users. In doing that, integration with Hive metadata and data are necessary. This proposal covers only Hive metadata (metastore) integration. (There will be a separate document covering data integration aspect (FLINK-10729).

Proposed Changes

There are two aspects of Hive metastore integration: 1. Make Hive's meta-object such as tables and views available to Flink and Flink is also able to create such meta-objects for and in Hive; 2. Make Flink's meta-objects (tables, views, and UDFs) persistent using Hive metastore as an persistent storage. This proposal covers both aspects.

The proposed changes are best demonstrated in the following class hierarchy:



`ReadableCatalog` Class

This class comes from renaming existing `ExternalCatalog` class. The reason for the change is that there isn't clear distinction between external and external as a an external catalog can also be used to store Flink's meta objects.

We need to adapt the existing APIs to accommodate other meta-objecs like tables and views that are present in Flink and also common in a typical database catalog.

```
trait ReadableCatalog {

  def open() : Unit
  def close() : Unit

  // Return all tables/views
  def listTables(): JList[String]
  def listViews(): JList[String]
  def getTable(val tableName: String) : CommonTable


  def getFunction(funcName: String) : UserDefinedFunction
  def listFunctions(): JList[String]
  def getSubCatalog(dbName: String): Catalog
  def listSubCatalogs(): JList[String]
}
```

`open()` and `close()` are added to `ReadableCatalog` interface and meant to take care of external connections. They might take some runtime context, but I leave it out for now.

~~`TableType` class~~

~~View is just a special type of table, with a query as its definition. Enum class `TableType` is used to differentiate different types of tables.~~

```
~~Enum TableType {
   TABLE, VIEW
}~~
```

~~The `TableType` enum allows for future expansion to specify other object types such as materialized view. It's adopted to cover both table and view.~~

`CommonTable` Class

Class `CommonTable` is a new class serves as the parent of table and view.

```
public abstract class CommonTable {
  Private Map<String, String> tableProperties;

  public CommonTable(Map<String, String> properties) {...}

  // Create a table source from the underlying table/view.
  // Null if table/view cannot be a source.
  TableSource toTableSource();
  ...
}
```

`CatalogTable` class

Class `CatalogTable` is renamed from `ExternalCatalogTable` with the following changes. It can be constructed either by a property map where everything about the table is encoded as key-value pairs (descriptor) or by POJO definition of a table (schema, stats, and additional properties). The former is likely used by Flink catalog where tables are defined as descriptors, and latter is more for tables from external catalog such as Hive metastore.

```
class CatalogTable extends CommonTable {
  private TableSchema tableSchema = null;
  Private TableStats tableStats = null;

  Public CatalogTable(Map<String, String]> properties) {
    super(properties)
  }

  Public CatalogTable(TableSchema tableSchema, TableStats tableStats,
Map<String, String]> properties) {
    super(properties)
    This.tableSchema= tableSchema;
    this.tableStats = tableStats;
  }

  // Create a table sink from this table (for writing)
  // Null if table cannot serve as a sink.
  TableSink toTableSink();
}
```

`CatalogView` class

```
class CatalogView extends CommonTable {
  // Original text of the view definition.
  Private String originalQuery;

  // Expanded text of the original view definition
  Private String expandedQuery;

  Public CatalogView(
     String originalQuery,
     String expandedQuery
     Map<String, String]> properties) {
     super(properties);
     this.originalQuery = originalQuery;
     this.expandedQuery = expandedQuery;
  }

}
```

`WritableCatalog` Class

This class comes from renaming `CrudExternalCatalog` class.

```
trait WritableCatalog extends ReadableCatalog {

  def createTable(tableName: String, table: CatalogTable, ignoreIfExists:
Boolean):unit
  def dropTable(tableName: String, ignoreIfNotExists: Boolean): Unit
  def alterTable(tableName: String, table: CatalogTable, ignoreIfNotExists: Boolean):
Unit

  def createView(viewName: String, view: CatalogView, ignoreIfExists: Boolean):unit
  def dropView(viewName: String, ignoreIfNotExists: Boolean): Unit
  def alterView(tableName: String, table: CatalogView, ignoreIfNotExists: Boolean):
Unit

  def createFunction(funcName: String, func: UserDefinedFunction, ignoreIfExists:
Boolean): Unit
  def dropFunction(funcName: String, ignoreIfNotExists: Boolean): Unit
  def alterFunction(funcName: String, func: UserDefinedFunction, ignoreIfNotExists:
Boolean): Unit

}
```

`FlinkInMemoryCatalog` Class

This class implements `WritableCatalog` and will be provided by Flink to host tables/views/functions defined in Flink, using either memory as storage.

While Flink metadata is currently stored in memory only, `FlinkInMemoryCatalog` encapsulates the metadata in a class conforming to catalog interface. This implementation will be enabled by default to be backward compatible. In such default configuration, this catalog will be also the default catalog. However, user might opt to a persistent implementation such as `FlinkHmsCatalog`(below).

`FlinkInMemoryCatalog` is similar to existing `InMemoryExternalCatalog` class, which exists only for testing purpose. We will recycle `InMemoryExternalCatalog` for `FlinkInMemoryCatalog`, and remove `InMemoryExternalCatalog` completely. Existing tests on `InMemoryExternalCatalog` will be migrated to `FlinkHmsCatalog`, which can further use an embedded Hive metastore.

`FlinkHmsCatalog` Class

For the catalog that will be used by Flink, we have to consider temporary meta objects such as tables and functions. Temporary meta objects are not persisted in a catalog. Rather, they are stored in memory only and associated with user sessions. When users disconnects, temporary meta objects are automatically removed. Otherwise, meta objects behaves like their regular counterparts. For instance, they share the same namespace hierarchy.

Temporary function isn’t supported in Flink yet, but temporary tables may be created via existing Flink Table API. Ideally, we should have some in-memory data structures associated with a user session. A in-memory catalog fits well for this purpose. Thus, we can leverage FlinkInMemoryCatalog for temporary meta objects in Flink.

Since Flink SQL doesn’t have a server-client architecture, there are no user sessions. In other words, there is only one single user session. In this case, where to hold the in-memory catalog doesn’t matter much. For now, we can just keep the in-memory catalog in `FlinkHmsCatalog` itself. (This might be moved to a session state object when user session is introduced.)

Thus, `FlinkHmsCatalog` class needs to extend `CatalogBase` in order to support temporary tables and its implementation is built in Flink but using Hive metastore as storage.

```
class FlinkHmsCatalog extends WritableCatalog {

  // An in-memory catalog instance own by this object to store temporary objects such
  // as temp tables and UDFs. It has the same namespace as the enclosing catalog.
  Private FlinkInMemoryCatalog memCatalog = new FlinkInMemoryCatalog();

  Public void createTempTable(String name, CommonTable table);
  Public void createTempFunction(String name, UserDefinedFunction function);
  ...
}
```

As such, `FlinkHmsCatalog` covers the second aspect of Flink-Hive metastore integration. The implementation has a reference to a `HiveMetastoreClient`, which basically represents a connection to a Hive metastore and encapsulates operations needed by Flink.

User can enable `FlinkHmsCatalog` by overwriting the default configuration. This can be done via table APIs such as `TableEnvironment.registerCatalog(“flink-hms”, new FlinkHmsCatalog())` and `TableEnvironment.setDefaultCatalog(“flink-hms”)`. Once this is done, `FlinkHmsCatalog` will replace the default `FlinkInMemoryCatalog` to store user defined tables, views , and UDFs.

While being provided out of box, `FlinkHmsCatalog` will reside in a the Hive connector package flink-connectors/flink-connector-hive to avoid dependency pollution.

`HiveCatalog` Class

It’s an implementation of `Writableatalog` to access Hive native tables, views, functions, etc. Similar to `FlinkHmsCatalog`, it has a reference to `HiveMetastoreClient`.

Conversion between a table descriptor of a Hive table and a live Hive table object is needed as `HiveMetastoreClient` only understand Hive object (but not the descriptor). During the conversion, validation is necessary and can be performed by the Hive connector (schema, format, and connector validators). While being related to this effort, Hive connector related development (factory, validator, etc) is out of the scope of this design doc.

Here is the outline for the conversion between Flink table descriptor and Hive table

1. Schema properties -> Hive table schema
2. Other properties -> Hive table properties
3. Stats -> Hive table stats

This class will also stay in the connector package.

`CatalogManager` Class

It manages all the registered `ReadableCatalog` instances. It also has a concept of default catalog, which will be selected when a catalog name isn't given in a meta-object reference.

```
public class CatalogManager {

       // the catalog to hold all registered and translated tables
       // we disable caching here to prevent side effects
       private val internalSchema: CalciteSchema =
CalciteSchema.createRootSchema(true, false)
       private val rootSchema: SchemaPlus = internalSchema.plus()

       // A list of named catalogs.
       private Map<String, ReadableCatalog> catalogs;
      // The name of the default catalog
       private String defaultCatalog = null;

      public CatalogManager(Map<String, ReadableCatalog> catalogs, String
defaultCatalog) {
        // make sure that defaultCatalog is in catalogs.keySet().
        This.catalogs = catalogs;
        this.defaultCatalog = defaultCatalog;
      }

       public void registerCatalog(String catalogName, ReadableCatalog catalog);

       Public void setDefaultCatalog(String catName);
       public ReadableCatalog getDefaultCatalog();
\
       public ReadableCatalog getCatalog(String catalogName) { return
catalogs.get(catalogName); }
       public List<String> getCatalogs();
}
```

`CatalogManger` will encapsulate Calcite's schema framework such that no code outside `CatalogManager` needs to interact with Calcite's schema except the parser which needs all catalogs. (All catalogs will be added to Calcite schema so that all external tables and tables can be resolved by Calcite during query parsing and analysis.)

~~`CatalogManager` can also keep track of any non-permanent meta objects such as temporary tables or functions.~~

YAML Configuration for Catalogs

The following is a demonstration of configurations for catalogs in Flink. Available catalog types are: `flink-in-memory`, `flink-hms`, and `hive`. Note that there should be at most one of "flink-hms" and "flink-inmemory". However, there could be multiple catalogs of type "hive".

```
catalogs:
   name: hive1
   Catalog:
     Type: hive
     connector:
       hive.metastore.uris:
"thrift://host1:10000,thrift://host2:10000"
       hive.metastore.username: "flink"
   name: flink
   Catalog:
     type:flink-hms
     Connector:
       hive.metastore.uris:
"thrift://host1:10000,thrift://host2:10000"
       hive.metastore.username: "flink"
       Hive.metastore.db: flink
   name: flink-builtin
   Catalog:
     Type: hive-in-memory
default-catalog: flink
```

We can have something similar to TableFactory utilizing Java Service Provider interfaces (SPI) for catalog discovering and creation.

```
public interface CatalogFactory {
   Map<String, String> requiredContext();
   List<String> supportedProperties();
}

public class FlinkInMemoryCatalogFactory implements CatalogFactory {
   Map<String, String> requiredContext() {
       val context = new util.HashMap[String, String]()
       context.put(CATALOG_TYPE, "flink-in-memory")
   }

   List<String> supportedProperties() {}
}

public class FlinkHmsCatalogFactory implements CatalogFactory {
   Map<String, String> requiredContext() {
       val context = new util.HashMap[String, String]()
       context.put(CATALOG_TYPE, "flink-hms");
   }

   List<String> supportedProperties() {
       Hive connection params, db to be used.
   }
}

public class HiveCatalogFactory implements CatalogFactory {
   Map<String, String> requiredContext() {
       val context = new util.HashMap[String, String]()
       context.put(CATALOG_TYPE, "hive");
   }

   List<String> supportedProperties() {
       Hive connection params
   }
}
```

We can put the three factory classes in `META_INF/services` file for auto discovery.

`TableEnvironment` Class

This is the existing class in table API, which will have a reference to `CatalogManager` instance that is to be added to replace the in-memory meta-objects and registered catalogs.

```
abstract class TableEnvironment(val config: TableConfig) {
…
  Private val catalogManager: CatalogManager;

  def registerCatalog(name: String, catalog: ReadableCatalog): Unit
  def setDefaultCatalog(catName: String);
  Def setDefaultDatabase(catName: String, dbName: String): unit
}
```

`TableEnvironment` class currently has a few `registerTable` methods taking different table definitions, such as `TableSource`, `TableSink`, Table (`RelTable`), and Table `InlineTable`.
Not every table that's such registered requires an external persistency. In fact, some of those tables are temporary (like `InlineTable`) or generally difficult to serialize (like `RelTable`). Thus, we need to identify cases where storing in a catalog is both needed and feasible. For such cases, we may need to either provide new APIs or adapt an existing APIs. The following are the two cases we identified and planned to support cataloging.

Table registered as `TableSource` or `TableSink`

`TableSource` and `TableSink` are likely created from descriptor properties, though not always. We plan to support cataloging only for sources or sinks that are created from a descriptor. To do that, we introduce a `Describable` interface.

```
trait Descriptable[T] {
  def getProperties: java.util.Map
}
```

With this, `TableEnvironment` is able to tell if a `TableSource` or `TableSink` is something that can be put in a catalog. For example:

```
  def registerTableSource(name: String, tableSource: TableSource[_]):
Unit = {
      checkValidTableName(name)
      if(tableSource instanceof Desciptable) {
         Register table with its descriptor properties
         ...
      } else {
      registerTableSourceInternal(name, tableSource)
      }
  }
```

For Flink built-in sources or sinks, such as `OrcTableSource, CsvTableSource` and `CsvTableSink`, we can let them implement Describable API. For example:

```
public class OrcTableSource
    implements BatchTableSource<Row>, ProjectableTableSource<Row>,
FilterableTableSource<Row>, Describable<Row> {
}
```

For `TableSource` or `TableSink` that doesn't implement `Describable` interface, then Flink will not persist them in a catalog.

In the future, Flink can further check if the source or sink implements serializable interface and if so store it in catalog with its serialized bytes. However, this is not included in the current plan.

Virtual Table

The second class of registered tables that should be stored in a catalog are essentially views, which is tables created from given queries. Such tables are currently registered as `RelTable`s. However, not every RelTable is such created.

To differentiate a view from other types of `RelTables`, we introduce a new API in `TableEnvironment` to allow user to register a view.

```
  def registerView(name: String, query: String): Unit = {
      checkValidTableName(name)
      Create a CatalogView and store it in flink catalog.
  }
```

Other Table Types

For other types of `RelTable`, in addition to `InlineTable`, `TableSource/Sink` that doesn't implement `Describable` interface, and any `Table` in general that do not fall into either of the two cases above, Flink will not store them in a catalog. Those tables are usually live only for the life time of the user session and they don't need to be persisted. They behave conceptually as temporary tables and will be stored in memory only.

Presently, such temporary tables are directly put in Calcite's schema. As we are introducing an in-memory catalog for temporary tables and the catalog interface only accept a `CommonTable`, we need to adapt those classes to `CommonTable` interface. The following class plays such a role.

```
public class TemporaryTable extends CommonTable {
    Private Table table;
    Public TemporaryTable(Table table) {
        super(new HashMap<String, String>);
        this.table=table;
        …
    }
}
```

~~Tables that are registered as `RelTable` are essentially views. We store them as views, which are special kind of tables, in Flink catalog. This can be achieved with existing registerTable() method taking a CatalogView argument instead of existing `registerTable(name: String, table: Table)`.~~

At the same time, we should encourage user to use the right APIs when persistency is desirable.

Additional Notes

1. When the in-memory catalog, `FlinkInMemoryCatalog,` is configured by default, SQL client YAML configuration works as before and nothing needs to be done at SQL client.
2. SQL client can use YAML file to configure the Flink catalog with `FlinkHmsCatalog`. SQL client attempts to register any table definitions in YAML file with "ignoreIfExist=true" so that user doesn't get errors or exceptions each time when their SQL client is processed. This is mostly for the backward compatibility purpose. User probably wants to use the client CLI instead to register tables than using the YAML file.
3. User can also register more external catalogs such as `HiveCatalog` by calling `TableEnvironment.registerCatalog("hive1", new HiveCatalog()).` For SQL client, the additional catalogs to be registered also comes from the YAML file.
4. With multiple catalog in one system, a table has to be identified by catalog name, database name, and table name. Thus, table reference needs to include catalog name, database name, and table name, such as `hive1.risk_db.user_events`. When catalog name is missing, it's assumed to mean the default catalog (whatever a catalog that is set as default).
5. We may also introduce default database concept in Flink. This corresponds to SQL "use xxx" where a database (schema) is set as the current one and any table without database/schema prefix is referring to the default database. Since Flink has multiple catalogs, the syntax would be "use cat1.db1", with which cat1 will be the default catalog and db1 will be the default database.
6. User may programmatically set or change the default catalog at any time by calling `TableEnvironment.setDefaultCatalog(String catName)` or set the default database by calling `TableEnvironment.setDefaultDatabase(String catName, String dbName).`
7. Hive metastore configuration will be discovered from the client host environment, though this behavior can be overwritten by explicitly provide Hive configuration.

`FlinkHmsCatalog` Design

Via `FlinkHmsCatalog` class, Flink stores its metadata in Hive metastore. In this case, Hive metastore is purely used as a metadata storage. While those objects are stored in Hive metastore, Hive (and other tools working with Hive metastore) may not fully understand those objects. This is expected. This is in contrast to Hive metastore as an external catalog where tables, along with other objects, created by Flink will conform to Hive standard, and other Hive compatible tools will be able to make sense of them.

Conversion between a table descriptor of a Flink table and a live Hive table object is needed as `HiveMetastoreClient` only understand Hive object (but not the descriptor). The same is true for views and UDFs. Here is the outline for the conversion:

4. Schema properties -> Hive table schema
5. Other properties -> table properties
6. Additional table properties: application.name=flink

`FlinkHmsCatalog` will be configured with a dedicated database (configurable, “flink” by default) in Hive as the namespace for all objects that Flink stores. To differentiate those objects created by Flink, there may also add a property called “application.name”, for which “Flink” will be explicitly set for those objects (if applicable).

This class will stay in the connector package.

Flink Table

Flink tables will be mapped to Hive table in Hive metastore. Since Flink meta objects are defined as descriptors (string properties), Hive table properties will be used extensively to store the descriptors. However, Flink table schema will be converted to Hive table schema.

Flink Views

In Flink, views are special type of tables with a SQL query which is the view definition (SQL statement). Thus, Flink views will be mapped as tables in Hive metastore.

Flink UDFs

Similarly, Flink UDFs will be mapped as UDF objects in Hive metastore. Unlike tables/views, Hive metastore doesn’t allow an UDF to carry a list of custom properties. For this, we may have to add such support in Hive.

`HiveMetastoreClient` Design

This just a boiler place to encapsulate a Hive metastore client and a bunch of methods tailored to the need of Flink.

```
Class HiveMetastoreClient {
  private final HiveConf hiveConf;
  private IMetaStoreClient metaStoreClient;

  Public void createTable(Table hiveTable);
  Public void dropTable(String name);
  …
}
```